2

WSRL-TN-46/89

AR-005-938

AD-A236 027

A PROCEDURE FOR GENERATING RUN-SYNCHRONISED DISJOINT STREAMS OF PSEUDO-RANDOM NUMBERS FOR USE IN SIMULATION

M.L. SCHOLZ

COMBAT SYSTEMS DIVISION
WEAPONS SYSTEMS RESEARCH LABORATORY

Approved for Public Release.

SEPTEMBER 1990

DSTO AUSTRALIA
DEPARTMENT OF DEFENCE
DEFENCE SCIENCE AND TECHNOLOGY ORGANISATION

DSTO AUSTRALIA

TECHNICAL NOTE
WSRL-TN-46/89

A PROCEDURE FOR GENERATING RUN-SYNCHRONISED DISJOINT STREAMS OF PSEUDO-RANDOM NUMBERS FOR USE IN SIMULATION

M.L. Scholz

ABSTRACT(U)

A procedure for generating disjoint streams of pseudo-random numbers for use in discrete event simulation is presented. The procedure facilitates the synchronisation of variates from run to run and can be employed in conjunction with several variance reduction techniques. It is based on Schrage's implementation of the Lehmer generator and includes enhancements to remove the restriction on usable multiplier values. Seeds are efficiently computed by a routine that exploits the number-theoretic properties of congruences. Machine-portable Fortran routines are listed for use on machines capable of performing 31-bit signed integer arithmetic.

Author's address:
Combat Systems Division
Weapons Systems Research Laboratory
PO Box 1700, Salisbury
South Australia

Requests to: Chief, Combat Systems Division

TABLE OF CONTENTS

1. INTRODUCTION

Disjoint *streams* (viz sequences) of pseudo-random numbers are frequently employed when the performances of several systems, system configurations, or operational policies are compared using discrete event simulation. Disjoint streams are employed in conjunction with several variance techniques, including Common Random Numbers and Antithetic Variates(ref.1), where the objective is to extract in the minimum possible time (or run length) confidence interval estimates for model response variables. These techniques require a generator that can produce independent streams and reproduce (viz synchronise) them from run to run.

During the course of building a GPSS/H simulation model of a local area network(ref.10) the author found it necessary to invoke an external random number generator for sampling continuous multi-parametric distributions, a facility lacking in GPSS/H. There is also a problem with the high-order serial correlation properties of the Tausworthe random number generator incorporated in release 0.99 of GPSS/H (reference 1, p 202).

The LCG, or Lehmer generator(ref.8), is the traditional source of pseudo-random numbers. The prime modulus multiplicative LCG (PMMLCG), described by Hutchinson(ref.4), is one of the most popular forms of the LCG and is noted for its simplicity. The statistical properties of the PMMLCG with modulus $2^{31}-1$ are the also best known.

The PMMLCG produces a sequence of non-negative integers $<X_1,X_2,......>$ from a specified initial non-negative integer, X_0, using the recursive linear congruence formula

$$X_{n+1} = (MX_n) \bmod N, \text{ for } n = 0,1,... \qquad (1)$$

where the *multiplier* (M) and the *modulus* (N) are integers and N is prime. The symbol "mod N" denotes the remainder obtained from division by N (see Appendix I, Definition 1). Provided $M \neq N$ and $X_0 \neq 0 \pmod N$, the values of the generated integers will lie in the closed interval [1,N-1]. The Lehmer sequence is cyclical with a period dependent on M. In the particular case where the multiplier is a primitive root of the modulus, that is, if the multiplier satisfies $M^{N-1} = 1 \pmod N$ and $M^t \neq 1 \pmod N$ for $0<t<N-1$, the PMMLCG has full period (N-1) and produces (N-1) unique variates in the cycle. A stream of real numbers $<Z_1,Z_2,....>$, whose values are asymptotically uniformly distributed in the open interval (0,1), is then simply obtained by scaling, viz $Z_n = X_n/N$. Variates with other distributions may be obtained by transformation (see reference 1 pp 306 to 347 and reference 7 pp 240 to 278).

The Lehmer generator is featured in many of the standard mathematical subroutine libraries (eg GGUBT in IMSL(ref.15), G05CAF in NAG(ref.13) and MTH$RANDOM in VAX/VMS(ref.16)). Recent studies by Fishman and Moore(ref.2) suggest that the statistical properties of existing PMMLCG implementations may not be optimal. They recommend the use of alternative multipliers.

An efficient, machine-portable computer program capable of generating synchronised disjoint streams of pseudo-random numbers on 32-bit machine architectures is described in this paper. It is based on the well-known Schrage algorithm(ref.11), a Fortran program which implements the PMMLCG very efficiently on 32-bit machines by using integer arithmetic. Improvements have been made to increase the numerical precision of the calculations to accommodate the multiplier values recommended by Fishman and Moore.

The remainder of this paper is organised as follows. In Section 2 alternative ways of allocating random numbers to stochastic processes in discrete event simulation models are illustrated using a tandem queuing model. Generation methods based on interleaved and sequential sampling of the Lehmer sequence are discussed. The latter method forms the basis of the two algorithms subsequently described in detail.

The first algorithm, described in Section 3, is the improved version of Schrage's random number generator. The second algorithm, described in Section 4, is used to initialise the seeds for the generator. Provided an upper bound can be specified on the number of required variates, the initialisation procedure will compute seeds that enable the random number generator to produce disjoint streams in accordance with the sequential sampling method.

The performance of each algorithm is assessed in Section 5. The speed of the improved Schrage algorithm is compared to the speed of the original Schrage algorithm and the IMSL GGUBT algorithm, and the time taken by the initialisation algorithm to compute the seeds is compared to the projected time to compute the seeds by the repeated application of the improved Schrage algorithm.

This research was undertaken with the sponsorship of the Director, Naval Combat Systems Engineering, Royal Australian Navy (Task No. NAV87/226).

2. GENERATION METHODS

Variates obtained from the Lehmer generator can be assigned to stochastic processes in a model from a single stream, but the variates may be correlated in the sense that if a process is modified in a subsequent simulation run, the sequence of variates assigned to other processes may be altered. Consequently this method cannot be used, except in the case of extremely simple models (eg single queues), in conjunction with the most popular of the variance reduction techniques (viz Common Random Numbers and Antithetic Variates). This is illustrated in the following example.

Consider a dual tandem queue. Denote the arrival of customer n by A_n, the commencement of service at service centre 1 by S_n and the commencement of service at service centre 2 by D_n, and suppose the following sequence of events occurs: $<A_1,S_1,A_2,D_1,S_2,A_3,D_2,S_3,A_4,A_5,S_4,\ldots>$. If numbers in the Lehmer sequence are chronologically assigned to processes and Z_1 is assigned to the first event (viz A_1), then customer inter-arrival times and centre 2 service times would be computed from $<Z_1,Z_3,Z_6,Z_9,Z_{10},\ldots>$ and $<Z_4,Z_7,\ldots>$ respectively. Now, consider a subsequent replication in which the service times of centre 1 are changed and the resulting event sequence is $<A_1,S_1,D_1,A_2,S_2,A_3,D_2,S_3,A_4,S_4,A_5,\ldots>$. Customer inter-arrival and centre 2 service times would be computed from $<Z_1,Z_4,Z_6,Z_9,Z_{11},\ldots>$ and $<Z_3,Z_7,\ldots>$ respectively, which are clearly different from before.

In order to assure that processes are independent from run to run, it is necessary to allocate a disjoint stream to each process. This may be achieved by interleaved sampling or by sequential sampling of the Lehmer sequence. In the previous example interleaved sampling involves allocating stream 1, comprising $<Z_1, Z_{1+k}, Z_{1+2k}, \ldots>$, to the arrival process; stream 2, comprising $<Z_2, Z_{2+k}, Z_{2+2k}, \ldots>$, to the centre 1 service process; and stream 3, comprising $<Z_3, Z_{3+k}, Z_{3+2k}, \ldots>$, to the centre 2 service process. Provided k is greater than or equal to the number of streams (*viz* $k \geq 3$), identical random arrival and centre 2 service times may be assigned to the respective customers in each run regardless of the distribution of centre 1 service times.

The main problem with interleaved sampling is that a variate Z_{i+nk} in stream i cannot be very efficiently generated from its progenitor $Z_{i+(n-1)k}$ because between one and a (k-1) intermediate variates, depending upon the method used. has to be generated (and wasted) to obtain each variate.

Sequential sampling is the best solution for generating disjoint streams because only one application of the PMMLCG congruence formula (equation (1)) is needed to generate each variate. In this case variates are allocated in contiguous subsequences as follows. Stream 1 comprises $<Z_1, Z_2, Z_3, \ldots, Z_j>$, stream 2 comprises $<Z_{j+1}, Z_{j+2}, \ldots, Z_k>$ and stream 3 comprises $<Z_{k+1}, Z_{k+2}, \ldots, Z_m>$. Provided not more than j, k-j and m-k variates are respectively required from streams 1, 2, and 3 during the simulation runs, the streams will be disjoint.

The initial integer variates, or *seeds* (Z_0, Z_j and Z_k), must be computed to ensure that the streams are disjoint. The computations need only be performed once prior to the first run. The computational problem is similar to that encountered previously in the interleaved sampling method, but it is more constrained because it is infeasible to use the PMMLCG formula when many thousands, or possibly millions, of intermediate variates must be computed.

3. THE IMPROVED SCHRAGE ALGORITHM

The naive implementation of equation (1) using P-bit precision positive integers will incur overflow unless the product $M(N-1) \leq 2^P-1$. Many generators of interest do not satisfy this criterion. A simple solution is to employ multiple-precision floating point variables, rather than integers, provided the precision of the mantissa is adequate. (On a given machine, numbers can usually be stored with higher precision in floating point representation.) However, a more efficient solution may be achieved using multiple-precision integer arithmetic where each variate and the constant multiplier is expressed as an expansion in the powers of an integer constant, $r = 2^k$, such that $k<P$. The modulo N product may then be obtained by manipulating the expansion coefficients.

The algorithm described by Schrage employs radix 2^{16} integer arithmetic to solve equation (1) on a 32-bit machine (*viz* P = 31) using a 31-bit modulus, $N = 2^{31}-1$, and a 15-bit multiplier, $M = 7^5 = 16807$. Note that although the Schrage algorithm employs a primitive root multiplier, any other multiplier will also work, provided it has less than 16-bit precision (*viz* a value less than 32 768).

Following recent studies by Fishman and Moore(ref.2) there has been considerable interest in the use of larger multipliers. These authors exhaustively tested the statistical properties of the PMMLCG with $N = 2^{31}-1$ for every possible primitive root multiplier and published a list of the 414 best multipliers. All of the multipliers exceed 23 bits in precision and thus cannot be employed in the original Schrage algorithm. Some of the listed multipliers in current usage include M = 630,360,016 in the SIMSCRIPT II simulation language(ref.5), M = 397,204,094 in the IMSL Fortran subroutine library(ref.15) and SAS language(ref.14), and M = 742,938,285 in the GPSS/H simulation language(ref.3).

The improved Schrage algorithm, described below, is similar to the original algorithm except for modifications that enable the use of 31-bit precision multipliers.

Expanding upon the original Schrage procedure, the integer variable (X_n) and the multiplier (M) in equation (1) are represented as quadratic and linear polynomials in the radix $r = 2^{15}$, viz

$$X_n = u_n^{(2)} r^2 + u_n^{(1)} r + u_n^{(0)}$$

and

$$M = v_n^{(1)} r + v_n^{(0)}.$$

The integers $u_n^{(0)}$, $u_n^{(1)}$, and $u_n^{(2)}$, (corresponding respectively to the 15 low order bits, the 15 middle order bits, and the high order bit of X_n) and $v^{(0)}$ and $v^{(1)}$ (corresponding respectively to the 15 low order bits and the 16 high order bits of M) are given by:

$$u_n^{(0)} = (X_n \bmod r^2) \bmod r,$$

$$u_n^{(1)} = \lfloor (X_n \bmod r^2) . r^{-1} \rfloor,$$

$$u_n^{(2)} = \lfloor X_n r^{-2} \rfloor,$$

$$v^{(0)} = M \bmod r,$$

and

$$v^{(1)} = \lfloor M r^{-1} \rfloor.$$

where $\lfloor \bullet \rfloor$ represents the largest integer smaller than, or equal to ($\bullet$) (see Appendix I, Definition 2).

The product (MX_n) may also be expanded as a polynomial of degree four in r:

$$MX_n = w_n^{(4)} r^4 + w_n^{(3)} r^3 + w_n^{(2)} r^2 + w_n^{(1)} r + w_n^{(0)} \qquad (2)$$

where the coefficients $w_n^{(0)},\ldots\ldots,w_n^{(4)}$, correspond respectively to the 15 low order bits, the three 15-bit groupings of the middle order bits, and the two high order bits of the 62-bit product. Each of the coefficients can be efficiently computed from the coefficients of the integer variable (X_n) and the multiplier (M) using the classical multiplication algorithm described by Knuth (reference 6, p 253).

Simulated division, described by Payne et al(ref.9), is then employed to compute the product, modulo N. Letting

$$\xi_n = \lfloor MX_n . r^{-2}/2 \rfloor = w_n^{(4)}.r^2/2 + w_n^{(3)}.r/2 + \lfloor w_n^{(2)}/2 \rfloor \qquad (3)$$

the variable, X_{n+1}, is obtained from

$$X_{n+1} = \begin{cases} Y_n \text{ if } Y_n > 0 \\ Y_n + N \text{ if } Y_n < 0 \end{cases}$$

where

$$Y_n = MX_n - N.(\xi_n+1). \qquad (4)$$

Substituting equations (2) and (3) into equation (4), and collecting terms in the powers of r, Y_n is then also obtained in terms of the expansion coefficients of the product, viz

$$Y_n = (w_n^{(4)} + 2w_n^{(2)} - 4(1 + \lfloor w_n^{(2)}/2 \rfloor)).(r^2/2)$$
$$+ (w_n^{(3)} + 2w_n^{(1)}).(r/2) + (1 + \lfloor w_n^{(2)}/2 \rfloor + w_n^{(0)}).$$

Note that each of the bracketed terms in the above expression may be computed and summed without overflow because their values lie within the range of 31-bit signed integer representation, viz $[-2^{31}+1,2^{31}-1]$.

The Fortran function DRANX, described in Appendix II, is an implementation of the improved Schrage algorithm. Note that Knuth's notation has been retained for the expansion coefficients in the Fortran program. The coefficients are therefore indexed in reverse order, commencing at one instead of zero.

4. INITIALISATION PROCEDURE

The number of iterations of equation (1) needed to obtain a seed from a *reference* variate X_0, in the Lehmer cycle is called the *offset*, $\theta(s)$, of stream s (s = 1,2,...,S). Assuming the offsets are totally ordered such that $0 \le \theta(1) < \theta(2) < \ldots < \theta(S)$, it is only necessary that the number of variates in any stream s does not exceed $(\theta(s+1)-\theta(s))$ to guarantee that the streams will be disjoint (viz sequences of variates do not overlap).

The period of any primitive root PMMLCG with modulus $N = 2^{31}-1$ should provide a sufficient number of streams of the required length for practical purposes. These generators can be configured to produce, for example, more than a thousand streams with one million variates per stream, or more than a hundred streams with ten million variates per stream. Generators with non-prime multipliers can also be employed but, because their cycle lengths are shorter, larger offsets must be specified to obtain the same number of variates per stream.

The Lehmer generator itself cannot be used to compute seeds because it is far too inefficient to compute each of the $(\theta(s)-1)$ intervening variates.

It can be shown by induction (Appendix I, Theorem 1) that equation (1) is equivalent to

$$X_n = (M^n X_0) \bmod N$$

so that a seed is given by:

$$X_0(s) = (M^{\theta(s)} X_0) \bmod N \qquad (5)$$

Equation (5) suggests a single step method for computing seeds but it is impractical because the powers of large multipliers cannot be efficiently computed or stored.

A new algorithm which exploits the number-theoretic properties of congruences to compute the seed of each random number stream is described below. It involves the relatively simple computation of a limited number of intervening variates. The Fortran subroutine STREAM, described in Appendix II, is based on this algorithm.

Firstly, it can be shown (Appendix I, Theorem 2 and Corollary 1) that:

$$X_{2n} = (Q.(X_n^2) \bmod N) \bmod N , \qquad (6)$$

where the factor Q is the integer solution of the Diophantine equation:

$$N.L + Q.X_0 = 1 \qquad (7)$$

and L is also an integer solution. Provided that N and X_0 are relatively prime (ie, the greatest common divisor of N and X_0 is unity), a numerical solution of equation (7) can be obtained using an extension to the classical Euclid algorithm described by Knuth (reference 6, p 352). Note that if Q is negative, N must be added to it prior to substitution into equation (6) in order to maintain X_{2n} non-negative.

Next, a unique sequence $A(k_0)$ can be defined on the set {0,1} for each integer, k_0:

$$A(k_0) = \langle a_i = 1-(k_i \bmod 2) : k_{i+1} = 1+(k_i-2)/2^{a_i}, k_{i+1} > 1, i=0,1,..,\ell-1 \rangle \qquad (8)$$

The elements of the sequence will be referred to as (binary) indicators. The number of indicators, ℓ, which is a function of k_0, is a maximum when (k_0+1) is a second or higher power of two; ie, if $k_0 = (2^P-1)$, then $\ell=2(P-1)$. Thus, ℓ will always satisfy $\ell \leq 2(\log_2(k_0+1)-1)$ provided $k_0>1$. A maximum of 60 indicators is required, for example, to represent a 31-bit integer (viz $P = 31$).

Now, by letting $k_0 = \theta(s)$, equation (5) may be resolved into a set of $(\ell+1)$ equations. Each equation is a congruence relation involving an indicator of $\theta(s)$, as follows:

$$X_{k_\ell} = (M^{a_\ell+1} X_0) \bmod N \tag{9}$$

$$X_{k_{\ell-1}} = (M^{a_{\ell-1}+1} X_{k_\ell}) \bmod N \tag{10.1}$$

$$\vdots$$

$$X_0(s) = X_{k_0} = (M^{a_0+1} X_{k_1}) \bmod N \tag{10.ℓ}$$

where $a_\ell \equiv 0$ (and $k_\ell \equiv 1$), and $k_0 > k_1 \ldots > k_\ell$. Note that equation (9) may be simplified to

$$X_{k_\ell} = (MX_0) \bmod N = X_1, \tag{11}$$

where X_1 is the first variate in the sequence generated by equation (1). Equations (9) to (10.ℓ-1) represent the ℓ intervening variates that must be computed between the reference variate and the seed.

When combined with equations (1) and (6), equations (10.1) to (10.ℓ) may be expressed by a single recursion formula,

$$X_i = (M^{1-a_i} Q^{a_i} (X_{i+1})^{1+a_i} \bmod N) \bmod N \tag{12}$$

where $i = \ell, \ell-1, \ldots, 1$ and $X_{\ell+1} \equiv X_1$ and $X_0(s) \equiv X_1$.

The seed $X_0(s)$ for stream s offset by $\theta(s)$ is therefore obtained from the following four step algorithm:

(1) compute the value of Q in equation (7) for the selected PMMLCG (and if $Q<0$ replace Q by Q+N);

(2) compute $X_{\ell+1} \equiv X_1$ from equation (11);

(3) determine the sequence of binary indicators $\langle a_0, a_1, \ldots, a_{\ell-1} \rangle$ corresponding to the integer $k_0 = \theta(s)$ using equation (8), and

(4) iteratively solve equation (12) starting with $i = \ell$. The final iterate (with $i = 0$) is the required seed.

5. COMPARATIVE SPEED OF PMMLCG ALGORITHMS

The mean execution times of the Fortran procedures DRANX (using the IMSL multiplier value) and RAND, which implement the improved and original Schrage random number generators respectively, were measured on a DEC VAX-8200 mainframe computer and compared to the execution time of the IMSL Fortran random number generator GGUBT(ref.15). GGUBT employs double-precision (64-bit mantissa) floating point arithmetic to solve equation (1) using

$$X_{n+1} = ([v^{(1)}.((rX_n) \bmod N)] \bmod N + [v^{(2)}X_n] \bmod N) \bmod N .$$

The advantage of using integer rather than floating point arithmetic is evident on comparing DRANX and GGUBT execution times. The penalty for increasing the precision of the multiplier in DRANX is reflected in the RAND and DRANX execution times. For each call DRANX took about half the time (202 ± 3 μs, 99% confidence interval (c.i.)) of GGUBT (411 ± 28 μs, 99% c.i.), whereas DRANX took about twice the time of RAND (119 ± 3 μs, 99% c.i.).

The total execution time of the initialisation procedure STREAM depends upon the number of streams required (S) and is a logarithmic function of the individual stream offsets. It is given by $860(\log_2[\prod_{1\le s\le S}(\theta(s)+1)]-S)$ (in μs) provided $\theta(s)\ge 1$ for all streams $s = 1,2,\ldots,S$. The constant of proportionality was determined by regression analysis of the execution times measured on the VAX. It takes into account that steps (1) and (2) of the algorithm need only be performed once, irrespective of the number of streams. The maximum execution time occurs with an offset $\theta(s) = N$ when the number of iterations of equation (12) is a maximum, viz $\ell = 60$. The effectiveness of the algorithm is apparent when the maximum execution time (26 ms) of STREAM (with S = 1) is compared with the estimated time taken (133 h) to generate the same seed from equation (1) using DRANX.

6. PORTABILITY

The programs DRANX and STREAM have been successfully tested on a variety of computers, operating systems and Fortran compilers. Further details are provided in Appendix II. A user may confirm the correct operation of the procedures by running the test program provided and comparing results. Although the programs were designed for use on 32-bit machines, they should work on any 16-bit machine (and possibly some 8-bit machines) provided the Fortran compiler that is used has a switch for INTEGER*4 format. The programs were shown to work on the 16-bit IBM PC/AT. Program execution times were not measured on the PC/AT but they are expected to be somewhat longer than on a 32-bit machine operating at the same clock speed due to the additional machine instructions needed to manipulate and store integer data. The execution time of DRANX on the 16-bit machine should be comparable to the execution time of the floating point GGUBT routine on the 32-bit machine.

7. SUMMARY

Disjoint streams of pseudo-random numbers are frequently employed when comparing the performances of several systems, system configurations, or operational policies using discrete event simulation. They are often used in conjunction with variance reduction techniques (eg Common Random Variates and Antithetic Variates) which require streams to be reproduced from run to run (viz synchronised between runs).

The allocation of disjoint streams to stochastic processes in a simulation model was illustrated for a simple tandem queue. Two methods of sampling the linear congruential (or Lehmer generator) sequence to obtain reproducible streams were described.

A Lehmer generator algorithm was described for generating run-synchronised disjoint streams on a computer capable of performing 31-bit signed integer arithmetic. The Schrage algorithm was selected because it is machine-independent and employs very efficient integer arithmetic. Although several prime-multiplicative LCG (PMMLCG) algorithms exist in mathematical subroutine libraries, none of them, to the author's knowledge, employs the optimal multiplier values recommended by Fishman and Moore. Modifications to the Schrage algorithm, which increase the multiplier precision from 24 to 31 bits, facilitate the use of the recommended values.

The algorithm was described mathematically and presented as a Fortran function (DRANX). The real variates derived from this function are asymptotically uniformly distributed in the unit interval (0,1) and can be transformed to any other common distribution.

The difficulties involved in computing the appropriate seeds for this generator were discussed. An efficient iterative procedure that exploits the number-theoretic properties of congruences to compute the seeds was described. It entails the conversion of a vector of stream offsets (denoting the positions of the seeds relative to a reference point in the Lehmer cycle) to seed values. This procedure was implemented as a Fortran subroutine (STREAM) and is used to initialise the random number generator. Provided the user-specified stream offsets satisfy a simple ordering relation, the streams can be guaranteed to be disjoint and reproducible between runs.

The execution times of the improved Schrage algorithm were measured and compared with the original Schrage and the IMSL GGUBT algorithms. Each call to GGUBT took about twice as long as a call to DRANX since GGUBT employs floating point arithmetic, whereas DRANX employs integer arithmetic. DRANX, on the other hand, took about twice as long as the original Schrage algorithm due to the additional complexity in using 31-bit precision multipliers. The maximum execution time of STREAM was measured. On the mainframe test computer STREAM took 26 ms against an estimated 133 h CPU time needed to generate a seed from a reference variate by repeated calls to DRANX. This result attests to the efficiency of the initialisation procedure. The portability of DRANX and STREAM have been verified on several machines (including a 16-bit machine), operating systems and Fortran compilers.

REFERENCES

No.	Author	Title
1	Bratley, P., Fox, B.L. and Schrage, L.E.	"A Guide to Simulation". 2nd Ed. Springer-Verlag, New York NY, 1987
2	Fishman, G.S. and Moore, L.R.	"An Exhaustive Analysis of Multiplicative Congruential Generators with Modulus $2^{31}-1$". SIAM J. Sci. Stat. Comput. Vol 7, No 1 pp.24-45, January 1986
3	Hendriksen, J.O. and Crain, R.C.	"GPSS/H User's Manual 2nd Edition". Wolverine Software Corp., Annandale VA., 1983
4	Hutchinson, D.W.	"A New Pseudorandom Number Generator". CACM Vol 9 pp.432-433, 1966
5	Kiviat, P., Villaneuva, R. and Markowitz, H.	"The SIMSCRIPT II Programming Language". Prentice-Hall, Englewood Cliffs, N.J., 1969
6	Knuth, D.E.	"The Art of Computer Programming Vol II - Semi-Numerical Algorithms". Addison-Wesley Press, Reading Mass., 1981
7	Law, A.M. and Kelton, W.D.	"Simulation Modelling and Analysis". McGraw-Hill, New York, 1982
8	Lehmer, D.H.	"Mathematical Methods in Large-Scale Computing Units". Ann. Comput. Lab. Vol 26, Harvard Univ. pp.141-146, 1951
9	Payne, W.H., Rabung, J.R. and Bogyo, T.P.	"Coding the Lehmer Pseudorandom Number Generator". CACM Vol 12, pp.85-86, 196
10	Scholz, M.L.	"Simulating Local Area Network Protocols with the General Purpose Simulation System (GPSS)". WSRL-TN-45/89, 1989
11	Schrage, L.	"A More Portable Fortran Random Number Generator". ACM Trans. Math. Softw. Vol 5, No 2 pp.132-138, 1979
12	Schriber, T.J.	"Simulation Using GPSS". John Wiley & Sons, New York, 1974

No.	Author	Title
13	-	"NAG User's Guide Mk 12". Vol 6, Numerical Algorithms Group Ltd, Oxford, 1987
14	-	"SAS (Statistical Analysis System) User's Guide: Basics, Version 5". SAS Institute Inc., Carey N.C., 1985
15	-	"The IMSL Library-Edition 9.2". Vol 2, IMSL Inc., Houston, Texas, 1984
16	-	"VMS 5.0 RTL Mathematics(MTH$) Manual". Digital Equipment Corporation USA, Order No. AA-LA72A-TE, April 1988

THIS IS A BLANK PAGE

APPENDIX I

NUMBER-THEORETIC DEFINITIONS AND PROOFS

Definition 1

Two integers a and b are *congruent modulo* m if their difference is a multiple of a non-zero integer m (m is called the *modulus*) viz., $(a-b) = nm$ for some integer, n. The congruence relation is expressed by the notation $a \equiv b \pmod{m}$.

Definition 2

$\lfloor \bullet \rfloor$ denotes the largest integer smaller than, or equal to a real parameter ($\bullet$) (viz, if $z = I+\delta$, where I is an integer and $0 \le \delta < 1$, then $\lfloor z \rfloor = I$).

Lemma 1

If a and m ($m \neq 0$) are integers, then $a \bmod m = a - m\lfloor a/m \rfloor$.

Proof: Let $b = a-m\lfloor a/m \rfloor$. Clearly the difference of a and b is an integer multiple of m, hence $b \equiv a \pmod{m} = a-m\lfloor a/m \rfloor$. QED.

Lemma 2

If the integers a and b are smaller than m, and $x = (ab) \bmod m$ and $y = ((a \bmod m).(b \bmod m)) \bmod m$, then $x = y$.

$$\text{Proof:}\quad y = ((a-m\lfloor a/m \rfloor).(b-m\lfloor b/m \rfloor)) \bmod m \text{ (by Lemma 1)}$$

$$= (ab-am\lfloor b/m \rfloor-bm\lfloor a/m \rfloor+m^2\lfloor a/m \rfloor.\lfloor b/m \rfloor) \bmod m$$

Since $a,b < m$, then $\lfloor a/m \rfloor = \lfloor b/m \rfloor = 0$ (by Definition 2).

Hence $y = (ab) \bmod m = x$. QED.

Theorem 1

The n-th term in the integer sequence $<X_j:j = 1,2,\ldots>$ generated by equation (1) $X_n = (M^nX_0) \bmod N$. This is a well-known result(ref.7, p 222) which can be proved by induction.

$$\text{Proof:}\quad X_1 = (MX_0) \bmod N$$

$$X_2 = (MX_1) \bmod N = (M(MX_0) \bmod N) \bmod N$$

$$= (M^2X_0) \bmod N \text{ (by Lemma 2)}$$

$$\vdots$$

$$\text{Assume } X_n = (M^nX_0) \bmod N$$

Consider

$$X_{n+1} = (MX_n) \bmod N = (M(M^nX_0) \bmod N) \bmod N$$

$$= (M^{n+1}X_0) \bmod N \text{ (again by Lemma 2)}$$

Hence, by induction,

$$X_n = (M^nX_0) \bmod N \text{ for all } n = 1,2,\ldots. \text{ QED.}$$

Lemma 3

If X_n and X_{2n} are the n-th and 2n-th terms of the integer sequence $<X_j:j=1,2,\ldots>$, generated by equation (1), then $(X_0X_{2n}) \bmod N = (X_n^2) \bmod N$.

Proof: $(X_0X_{2n}) \bmod N = (X_0(M^{2n}X_0) \bmod N) \bmod N$ (by Theorem 1)

$$= ((M^nX_0) \bmod N.(M^nX_0) \bmod N) \bmod N \text{ (by Lemma 2)}$$

$$= (X_n^2) \bmod N. \text{ QED.}$$

Theorem 2

If X_n and X_{2n} are terms of the integer sequence $<X_j:j=1,2,\ldots>$, generated by equation (1), then X_{2n} may be expressed as a function of X_0 and X_n by $X_{2n} = (Q(X_n^2) \bmod N) \bmod N$, where Q is the solution of the congruence relation $(QX_0) \bmod N = 1$.

Proof: Suppose $X_{2n} = (Q(X_n^2) \bmod N) \bmod N$

then $(X_0X_{2n}) \bmod N = (X_0(Q(X_n^2) \bmod N) \bmod N$

$$= ((QX_0) \bmod N.(X_n^2) \bmod N) \bmod N \text{ (by Lemma 2)}$$

$$= (X_n^2) \bmod N \text{ (by Lemma 3)}$$

The supposition is correct iff $(QX_0) \bmod N = 1$. QED.

Corollary 1: The solution Q of $(QX_0) \bmod N = 1$ is given by the solution of the linear Diophantine equation: $NL + QX_0 = 1$, where Q and L are integers.

Proof: By Definition 1, $QX_0 - 1 = -NL$, for any integer L.

Hence $NL + QX_0 = 1$. QED.

APPENDIX II

FORTRAN ROUTINES

In Sections II.1 and II.2 two ANSI Fortran 66/77 compatible procedures (STREAM and DRANX), which implement the multiple stream pseudo-random number generator, are described. Both algorithms are designed to be machine-portable and suitable for computers that can perform 31-bit signed integer arithmetic. The portability of the routines has been demonstrated using four different compilers and three machines/operating systems as follows:

Machine	Operating system	Compiler
IBM 3081	TSO/E 2.0 + MVS/XA 2.1.2	IBM Fortran/G1 1.1[1]
DEC VAX 8200	VMS 4.7	VAX Fortran 4.8-276[2] [3]
IBM PC/AT	DOS 3.1	Ryan-McFarland Fortran 1.0[1]
"	"	Ryan-McFarland Fortran 2.4[3]

[1] ANSI 66 [2] ANSI 66 using /NOF77 compiler option [3] ANSI 77

Disjoint streams are obtained by extracting the variates in each stream from different segments in the cycle of a multiplicative prime modulus linear congruential (Lehmer) generator with modulus $2^{31}-1$. The Fortran function DRANX, which generates the variates, is an improved version of Schrage's algorithm(ref.12). The numerical precision of the Lehmer multiplier has been increased to 31 bits to facilitate the use of the optimal multipliers recommended by Fishman and Moore(ref.2). Prior to calling the generator, the seeds for each stream are initialised by an efficient procedure (STREAM) that exploits the number-theoretic properties of congruences. The variates obtained from the generator are asymptotically uniformly distributed in the open interval (0,1).

A FORTRAN procedure, listed in Section II.3, demonstrates the method of invoking the generator. The test results are illustrated in Section II.4.

II.1 **INITIALISATION PROCEDURE (STREAM)**

```
C
C     SUBROUTINE STREAM IS CALLED ONCE PRIOR TO INVOKING THE FUNCTION
C     DRANX. STREAM OFFSETS ARE INPUT IN VECTOR IVEC WHICH ARE
C     CONVERTED TO SEEDS UPON RETURN.
C
C     ARGUMENT DESCRIPTIONS:
C
C       NS - THE MAXIMUM NUMBER OF STREAMS TO BE ESTABLISHED (INTEGER)
C
C       IVEC - VECTOR (INTEGER) OF DIMENSION NS USED TO INPUT STREAM
C              OFFSETS AND SUBSEQUENTLY TO STORE THE LAST VARIATES
C              GENERATED IN EACH STREAM.
C
C       IREF - THE REFERENCE VARIATE (INTEGER)
C
C       MULT - LEHMER MULTIPLIER (INTEGER)
C
C     COMMON BLOCKS: LABELLED COMMON BLOCK /MPARM/ USED INTERNALLY
C            TO PASS DATA FROM STREAM TO FUNCTION DRANX.
C
C       USAGE:
C
C           /MPARM/V1,V2
```

```
C
C       VARIABLES:
C
C           V1 - THE REMAINDER (INTEGER) FROM THE DIVISION
C                OF MULT BY 2**15
C
C           V2 - THE QUOTIENT (INTEGER) FROM THE DIVISION
C                OF MULT BY 2**15
C
C
C     OTHER ROUTINES CALLED: IQFAC AND LMOD
C
      SUBROUTINE STREAM(NS,IVEC,MULT,IREF)
      IMPLICIT INTEGER (A-Z)
      DIMENSION IVEC(NS),A(60)
      COMMON /MPARM/V1,V2
      DATA B15/32768/
C   .. DETERMINE EXPANSION COEFFS OF MULT FOR FUNCTION DRANX
      V1=MOD(MULT,B15)
      V2=MULT/B15
C   .. COMPUTE Q (STEP 1)
      Q=IQFAC(IREF)
C   .. COMPUTE X1 (STEP 2)
      X1=LMOD(IREF,MULT)
      DO 10 S=1,NS
C   .. DETERMINE INDICATORS A(1),..,A(L) FOR STREAM S (STEP 3)
        K0=IVEC(S)
        IF (K0.EQ.0) IVEC(S)=IREF
        I=1
20      IF (K0.LE.1) GOTO 30
          A(I)=1-MOD(K0,2)
          K0=1+(K0-2)/2**A(I)
          I=I+1
        GOTO 20
30      I=I-1
C   .. SOLVE EQUATION (18) FOR VARIATES (STEP 4)
        XHAT=X1
40      IF (I.LT.1) GOTO 50
          XHAT=LMOD(MULT**(1-A(I))*Q**A(I),
     X           LMOD(XHAT,XHAT**A(I)))
          I=I-1
        GOTO 40
50      IF (K0.NE.0) IVEC(S)=XHAT
10    CONTINUE
      RETURN
      END
C
C     FUNCTION IQFAC COMPUTES THE FACTOR Q IN EQ.(13) USING
C     EUCLID'S ALGORITHM (USING KNUTH'S NOTATION, SEE KNUTH P.352)
C
      FUNCTION IQFAC(IREF)
      IMPLICIT INTEGER (A-Z)
      U1=1
      U2=0
      U3=2147483647
      V1=0
      V2=1
      V3=IREF
10    IF (V3.EQ.0) GOTO 20
        Q=U3/V3
        T1=U1-V1*Q
        T2=U2-V2*Q
        T3=U3-V3*Q
        U1=V1
        U2=V2
        U3=V3
        V1=T1
        V2=T2
        V3=T3
      GOTO 10
20    IQFAC=U2
      IF (U2.LT.0)IQFAC=IQFAC+U
      RETURN
      END
C
C     FUNCTION LMOD COMPUTES (IX*IY) MOD (2**31-1), WHERE
C     IX,IY≤(2**31-1) USING CLASSICAL MULTIPLICATION ALGORITHM (USING
C     KNUTH'S NOTATION, SEE KNUTH P.253) AND SIMULATED DIVISION.
C
      FUNCTION LMOD(IX,IY)
      IMPLICIT INTEGER (A-Z)
      DATA B14/16384/,B15/32768/,B29/536870912/,B30/1073741824/
      DATA P/2147483647/
      IXS=IX
      U1=IXS/B30
      IXS=IXS-U1*B30
      U2=IXS/B15
      U3=IXS-U2*B15
      V1=IY/B15
      V2=IY-V1*B15
      T=U3*V2
      K=T/B15
      W5=T-K*B15
      T=U2*V2+K
      K=T/B15
      W4=T-K*B15
      T=U1*V2+K
      K=T/B15
      W3=T-K*B15
      W2=K
      T=U3*V1+W4
      K=T/B15
      W4=T-K*B15
      T=U2*V1+W3+K
      K=T/B15
      W3=T-K*B15
      T=U1*V1+W2+K
      K=T/B15
      W2=T-K*B15
      W1=K
      TX=1+W3/2
      LMOD=(W5+TX)+B14*(W2+2*W4)+B29*(W1+2*W3-4*TX)
      IF (LMOD.LT.0) LMOD=LMOD+P
      RETURN
      END
```

II.2 **RANDOM NUMBER GENERATOR (DRANX)**

```
C
C     FUNCTION DRANX RETURNS A DOUBLE PRECISION VARIATE UNIFORMLY
C     DISTRIBUTED IN THE OPEN INTERVAL (0,1)
C
C     ARGUMENT DESCRIPTION:
C
C
C       IVAR - INTEGER VARIATE (ALTERED BY THE ROUTINE)
C
C     COMMON BLOCKS: LABELLED COMMON BLOCK /MPARM/ USED INTERNALLY
C            TO PASS DATA FROM SUBROUTINE STREAM TO DRANX (FOR
C            DETAILED INFORMATION REFER TO SUBROUTINE STREAM).
C
C     OTHER ROUTINES CALLED: NONE
C
      FUNCTION DRANX(IVAR)
      IMPLICIT INTEGER (A-C,E-X)
      COMMON /MPARM/V1,V2
C  .. INITIALISE B14=2**14, B15=2**15, B29=2**29, B30=2**30, P=2**31-1
      DATA B14/16384/,B15/32768/,B29/536870912/,B30/1073741824/
      DATA P/2147483647/
C  .. PASS PARAMETERS FROM INITIALISING ROUTINE
C  .. COMPUTE EXPANSION COEFFICIENTS OF IVAR
      U1=IVAR/B30
      IVAR=IVAR-U1*B30
      U2=IVAR/B15
      U3=IVAR-U2*B15
C  .. COMPUTE PRODUCT OF MULT AND IVAR (KNUTH'S NOTATION)
      T=U3*V1
      K=T/B15
      W5=T-K*B15
      T=U2*V1+K
      K=T/B15
      W4=T-K*B15
      T=U1*V1+K
      K=T/B15
      W3=T-K*B15
      W2=K
      T=U3*V2+W4
      K=T/B15
      W4=T-K*B15
      T=U2*V2+W3+K
      K=T/B15
      W3=T-K*B15
      T=U1*V2+W2+K
      K=T/B15
      W2=T-K*B15
      W1=K
      TX=1+W3/2
      IVAR=(W5+TX)+B14*(W2+2*W4)+B29*(W1+2*W3-4*TX)
      IF (IVAR.LT.0) IVAR=IVAR+P
C  .. NORMALISE VARIATE
      DRANX=IVAR*4.6566128752457969D-10
      RETURN
      END
```

II.3 **TEST PROGRAM**

```
C     THIS PROGRAM DEMONSTRATES THE USAGE OF THE ROUTINES DRANX
C     AND STREAM. THE OUTPUT LISTS:
C
C     (1) THE SEEDS COMPUTED BY THE SUBROUTINE STREAM FROM THE
C         OFFSETS ASSIGNED TO STREAMS ONE TO FOUR AS FOLLOWS:
C
C             STREAM      OFFSET
C               1          99,999
C               2         999,999
C               3       9,999,999
C               4      99,999,999
C
C     (2) THE FIRST 10 SCALED (0,1) VARIATES IN EACH OF THE STREAMS
C         THAT ARE COMPUTED BY FUNCTION DRANX.
C
C     CONSTANT PARAMETERS:
C
C         MULT  - LEHMER MULTIPLER (GGUBS VALUE 397,204,094).
C         IREF -  REFERENCE VARIATE (GGUBS SEED 1,891,356,973).
C         IVEC(1),...,IVEC(4) - STREAM OFFSETS (SEE NOTE 1 ABOVE)
C
      INTEGER  MULT,IREF,IVEC(4),NS,S,LOOP
      DATA MULT/397204094/,IREF/1891356973/,NS/4/
      DATA IVEC/99999,999999,9999999,99999999/
      EXTERNAL DRANX,STREAM
C
C     .. INITIALISE 4 STREAMS
C
      CALL STREAM(NS,IVEC,MULT,IREF)
C
C     .. PRINT SEEDS
C
      WRITE (6,5)
5     FORMAT(' STREAM        SEED')
      WRITE (6,10)(S,IVEC(S),S=1,NS)
10    FORMAT(4X,I1,7X,I10)
C
C     .. PRINT VARIATES
C
      WRITE (6,15)
15    FORMAT(10X,/' RANDOM NUMBERS GENERATED BY DRANX'//
     X' POSITION   STREAM 1    STREAM 2    STREAM 3    STREAM 4')
      DO 30 LOOP=1,10
        WRITE(6,20)LOOP,(DRANX(IVEC(S)),S=1,NS)
20      FORMAT(3X,I2,6X,4(F9.7,3X))
30    CONTINUE
      END
```

II.4 **TEST RESULTS**

STREAM	SEED
1	1496156467
2	1851398463
3	81679943
4	287046138

RANDOM NUMBERS GENERATED BY DRANX

POSITION	STREAM 1	STREAM 2	STREAM 3	STREAM 4
1	0.9922353	0.4811183	0.1602005	0.3419817
2	0.4851381	0.1301162	0.9993262	0.3450323
3	0.2193673	0.7638627	0.5313926	0.9414775
4	0.8110999	0.3543463	0.5686302	0.3577577
5	0.8778924	0.1577095	0.0287373	0.7472422
6	0.1481317	0.5516356	0.6789374	0.8971976
7	0.3538447	0.6098807	0.9130332	0.6669795
8	0.5466737	0.4177297	0.6134635	0.6418861
9	0.3290880	0.5880992	0.2224111	0.9184812
10	0.6634960	0.3210178	0.1404637	0.5570008

DISTRIBUTION

	Copy No.
Defence Science and Technology Organisation	
Chief Defence Scientist First Assistant Secretary Science Policy Director General Science Resources Planning and Commercialisation Director General Science and Technology Programs Director General Space and Scientific Assessments Assistant Secretary Science Corporate Management Assistant Secretary Development Projects	} 1
Counsellor, Defence Science, London	Cnt Sht Only
Counsellor, Defence Science, Washington	Cnt Sht Only
Defence Science Representative, Bangkok	Cnt Sht Only
Scientific Adviser, Defence Research Centre, Kuala Lumpur	Cnt Sht Only
Scientific Advisor to Defence Central	1
Weapons Systems Research Laboratory	
Director,Weapons Systems Research Laboratoy	1
Chief, Combat Systems Division	1
Research Leader, Combat Systems	1
Head, Combat Systems Integration	1
Head, Combat Systems Effectiveness	1
J.G. Schapel, Combat Systems Integration	1
S.J. Miller, Combat Systems Integration	1
M.L. Scholz, Combat Systems Integration	3
R.D. Anderson, Systems Performance and Analysis	1
Libraries and Information Services	
Librarian, Technical Reports Centre, Defence Central Library, Campbell Park	1
Document Exchange Centre Defence Information Services for:	
Microfiche copying	1
United Kingdom, Defence Research Information Centre	2
United States, Defense Technical Information Center	2
Canada, Director Scientific Information Services	1

New Zealand, Ministry of Defence	1
National Library of Australia	1
Main Library, Defence Science and Technology Organisation, Salisbury	2
Library, DSD, Melbourne	1
Library, DSTO, Sydney	1
Library, Aeronautical Research Laboratory	1
Library, Materials Research Laboratory	1
Australian Defence Force Academy Library	1
British Library, Document Supply Centre	1
Department of Defence	
Director of Departmental Publications	1
Joint Intelligence Organisation (DSTI)	1
Navy Office	
Navy Scientific Adviser	Cnt Sht Only
Director, Naval Combat Systems Engineering	1
Other Departments and Organisations	
SA Institute of Technology	
Dr D.M. Panton, School of Mathematics and Computer Studies	1
Spares	5
Total number of copies	39

DOCUMENT CONTROL DATA SHEET

Security classification of this page : UNCLASSIFIED

1 DOCUMENT NUMBERS	
AR Number :	AR-005-938
Series Number :	WSRL-TN-46/89
Other Numbers :	

2 SECURITY CLASSIFICATION	
a. Complete Document :	Unclassified
b. Title in Isolation :	Unclassified
c. Summary in Isolation :	Unclassified

3 DOWNGRADING / DELIMITING INSTRUCTIONS

4 TITLE

A PROCEDURE FOR GENERATING RUN-SYNCHRONISED DISJOINT STREAMS OF PSEUDO-RANDOM NUMBERS FOR USE IN SIMULATION

5 PERSONAL AUTHOR (S)

M.L. Scholz

6 DOCUMENT DATE

September 1990

7 7.1 TOTAL NUMBER OF PAGES	21
7.2 NUMBER OF REFERENCES	16

8 8.1 CORPORATE AUTHOR (S)

Weapons Systems Research Laboratory

8.2 DOCUMENT SERIES and NUMBER
Technical Note
46/89

9 REFERENCE NUMBERS	
a. Task :	AIR 86/166
b. Sponsoring Agency :	DSTO

10 COST CODE

11 IMPRINT (Publishing organisation)

Defence Science and Technology Organisation

12 COMPUTER PROGRAM (S)
(Title (s) and language (s))

13 RELEASE LIMITATIONS (of the document)

Approved for Public Release.

Security classification of this page : UNCLASSIFIED

Security classification of this page : UNCLASSIFIED

14	ANNOUNCEMENT LIMITATIONS (of the information on these pages)
	No limitation

15	DESCRIPTORS		16 COSATI CODES
a. EJC Thesaurus Terms	Congreuences Number theory Simulation		1201
b. Non - Thesaurus Terms	Pseudo-random numbers		

17 SUMMARY OR ABSTRACT
(if this is security classified, the announcement of this report will be similarly classified)

(U) A procedure for generating disjoint streams of pseudo-random numbers for use in discrete event simulation is presented. The procedure facilitates the synchronisation of variates from run to run and can be employed in conjunction with several variance reduction techniques. It is based on Schrage's implementation of the Lehmer generator and includes enhancements to remove the restriction on usable multiplier values. Seeds are efficiently computed by a routine that exploits the number-theoretic properties of congruences. Machine-portable Fortran routines are listed for use on machines capable of performing 31-bit signed integer arithmetic.

Security classification of this page · UNCLASSIFIED